

AIARI Pascal
Documentation Directory

Release 0.0

June 2, 1980



MT MicroSYSTEMS

Specializing In Innovative
Information Processing

June 2, 1980

Dear Atari Pascal User:

This is Release 0.0 of ATARI Pascal. Enclosed is documentation describing how to use ATARI Pascal and a description of the language as implemented to-date.

NOTE: Release 0.0 is EXPERIMENTAL and is for
ATARI INTERNAL USE ONLY!!

For HELP (if really desperate) call:

Mike Lehman (714) 753-4856

or

Wink Saville (714) 942-2251



AIARI Pascal Release 0.0 Document Contents

Pascal Operator's Guide

- 1.0 Introduction
- 2.0 Monitor Operation
 - 2.1 Compiling
 - 2.2 Editing
 - 2.3 Syntax scan only
 - 2.4 Program execution
 - 2.5 ATARI DOS interaction
- 3.0 System Memory Map

AIARI Pascal 0.0 Language Status

- 1.0 Introduction
- 2.0 Pascal Standard
- 3.0 Atari Extensions

AIARI Pascal 0.0 Language User's Guide

- 1.0 Source Program Preparation
- 2.0 Compiler control toggles
- 3.0 Compiler Informational Output
- 4.0 Compile-time errors

AIARI Pascal Emulator Guide

- 1.0 Introduction
- 2.0 General Information
- 3.0 Calling machine code from Pascal routines

Sample AIARI Pascal Program Listings

- 1.0 PRIMES
- 2.0 TESTGET
- 3.0 TESTRDL
- 4.0 TESTGNB



AIARI Pascal

Operator's Guide

Release 0.0

June 2, 1980

Table of Contents

1.0	Introduction
2.0	Monitor Operation
2.1	Compiling
2.2	Editing
2.3	Syntax scan only
2.4	Program execution
2.5	ATARI DOS interaction
3.0	System Memory Map

1.0 Introduction

This document is designed to serve as a reference on the use of the ATARI Pascal system, Release 0.0 . This document does not teach the Pascal language and it does not teach the operation of the ATARI 800 computer and DOS. This document does teach the operation of the ATARI Pascal monitor, compiler and associated programs.

The overall operation of the ATARI Pascal system is controlled by the MONITOR program. The user will request compilation and execution via this program. In addition this program has been set up to handle loading of the ATARI developed editor in a special manner described in section 2.2. Before describing in detail the operation of the monitor and the compiler a few words of introduction are necessary.

Throughout this document the term <filename> is used to indicate a full ATARI device/filename combination (e.g. D:TESTPROG.PAS). Pascal source programs must have a three character extension (.PAS is recommended but not required). Output from the compiler Phase 0 (Syntax scanner) has the default extension .TOK (again suggested but not required). Output from the compiler Phase 2 (Code generator) has the default extension .COD (once more, suggested but not required).

The general rules for system operation are that when a default is available the user must type only the <return> key to accept the default (used in the MONITOR for file name prompting). Also, in the case of error, the system will give the user a choice of continuing or aborting. This is typically done with space being used for a signal to continue and the ESC key being used as a signal to abort. If the user chooses to abort any files open for output (e.g. a .TOK file in Phase 0 or a .COD file in Phase 2) will be deleted.

2.0 Monitor Operation

The control of the ATARI Pascal system is maintained by a monitor program (in the file called MONITOR on the distribution disk). This program will allow the user to call the compiler, call the editor, syntax scan, execute compiled programs and call the ATARI DOS. This section describes the user's initial introduction to the ATARI Pascal system as well as a detailed look at each individual command in the monitor program.

First, the contents of the distribution disk are as follows:

1. DOS.SYS - Standard ATARI DOS (9/24/79)
2. MONITOR - A Pascal program described here
3. PH0 - Phase 0 of the compiler
(performs syntax scanning and lexical analysis. outputs .TOK files)
4. PH1 - Phase 1 of the compiler
(builds symbol table in memory)
5. PH2 - Phase 2 of the compiler
(reads .TOK file and uses symbol table to generate .COD output file)
6. ERRORS.TXT- Text for standard error messages as found in Jensen and Wirth
7. PASCAL - Emulator and Monitor combined in a load and go program (will start automatically when Loaded via the DOS)
8. EMULATOR - 6502 Intermediate code Emulator (machine language program)

INITIAL OPERATION OF ATARI PASCAL

As a first step, the user must have the ATARI Pascal distribution diskette and the ATARI Pascal sample programs diskette. To take a test drive through ATARI Pascal perform the following steps:

1. Place Pascal sample programs disk in drive 2
2. Place Pascal system disk in drive 0
3. BOOT the ATARI 800 by turning power off and on
4. When the DOS menu appears type L
5. When asked LOAD FROM WHAT FILE? reply PASCAL and depress <return>
6. Wait about 40 seconds and then the ATARI Pascal display should appear and the prompt line should be displayed with the cursor at the end of the line.
7. Type C followed by <return> to begin a compilation
8. The message 'Source file name? ' should appear
9. Type D2:PRIMES.PAS and then <return>
10. The message 'Token file name? ' followed by '<return> for default:' should appear
11. Type <return> to use D2:PRIMES.TOK as the output file name from Phase 0 of the compiler
12. The message 'Code file name? ' followed by '<return> for default:' should appear
13. Type <return> to use D2:PRIMES.COD as the output file name from Phase 2 of the compiler
14. The message 'Loading Syntax Scanner' should appear
15. The compiler will now load a run (this will take approximately 2 minutes) and finally display: 'Compilation completed' and 'Type <return> to continue'.

<continued on next page>

16. The user should type <return> and control will be returned to the MONITOR program.
17. Now to run the program!. The user should type R followed by <return>
18. The message 'FILE NAME:' will appear
19. The user should type D2:PRIMES.COD and <return>
20. The program will be loaded and display:
'Type <return> to start'
21. The user should type <return> and wait about 27 seconds. (This takes 54 seconds on UCSD Pascal running on an Apple-II)
22. The program will print '1899 primes'
23. The user should type <return> and will then be returned to the DOS level.
24. To continue using Pascal the user should type L and then PASCAL and the monitor will be reloaded and executed.

2.1

Comments

This section is for the 'advanced' user. If you have not read the previous section you are highly advised to do so!

The ATARI Pascal compiler consists of three Phases: PH0, PH1, and PH2. These programs are written in ATARI Pascal which have been cross-compiled on a Z80 system running ATARI Pascal. The breakdown of functions for each phase is as follows:

PH0 - Read a .PAS file, syntax scans, converts to internal tokenized form and writes a .TOK file.

PH1 - Read a .TOK file and create a symbol table in memory.

PH2 - Read a .TOK file, generate P-code and write this P-code to a .COD file using the symbol table created in PH1.

The user's entry to the compiler is either at PH0 or at PH1 but in all cases from the MONITOR ONLY, DIRECT EXECUTION IS NOT ALLOWED. If the user enters a .TOK file name as the source file name in the monitor the system will ask the Code file name and begin execution directly at PH1. In all other cases the system will begin execution at PH0.

The user may specify each file name specifically or may take the defaults. The defaults retain the first part of the file names and changes the extensions to .TOK and .COD for the Token and Code file names respectively.

All files (in Release 0.0) are sequential and may exist on either disk or cassette with the exception of the .TOK file. This file must exist on disk because it is opened and read by both PH1 and PH2.

Execution of PH0 alone is described in section 2.3. This is necessary for INCLUDE files (see section 2.0 of the Language User's Guide for an explanation of INCLUDE files).

2.2

Editing

This section describes the MONITOR's interface with the text editing system being developed by ATARI. The MONITOR expects that the editor writer will create a file called MEDIT and it will contain a load file in the following format:

\$5300 - \$53FF Program to move \$5400..?? to location 3000 then JMP \$3000.

\$5400 - ???? Object code for editor to run at \$3000

This will undoubtedly change in the future.

The user may also use the editor in the Assembler/Editor cartridge but this will require reloading the entire system because of insertion and removal of the cartridge.

2.3 Syntax scan only

The user should read section 2.0 of the ATARI Pascal Language User's Guide for a discussion of INCLUDE files before reading this section.

PHO can be called to tokenize and syntax scan only without compilation. The user uses the S command from the MONITOR level and supplies only the input and token file names. When PHO is completed, either normally or on error, control will return to the MONITOR.

INCLUDE files must be tokenized before they are used in the compilation of a program. The user should use the S command to perform this function and create a .TOK file.

2.4

Pascal execution

Once a program has been compiled into P-code by the ATARI Pascal compiler the user may (MUST) run this program from the MONITOR. The user types R and <return> and then when prompted for FILE NAME the user should enter the name of the .COD file containing the program to run.

PROGRAMMING NOTES:

When a program completes the Emulator will exit via the warmstart vector in the ROM operating system. If the user wishes to return to the MONITOR the user must open the D:MONITOR file and use the CHAIN built-in procedure. See the MONITOR and compiler listings for examples.

When a program exits via the warmstart vector the screen will clear. The user should use a READLN statement at the end of the program and type <return> after the output from the program has been seen to prevent the output from being lost.

2.5

ATARI DOS interaction

The user may, from time to time, wish to exit to ATARI DOS and re-enter the monitor. The D)os command allows the user to exit to DOS. The use may then perform any command (except DUPLICATE FILE USING THE PROGRAM AREA) and return to the MONITOR via M <return> 5300 <return>. If the user destroys the program area the user should use L <return> PASCAL <return> to return to the MONITOR.

3.0 System Memory Map

Atari Pascal System Memory Map

0000 - 2A80	ATARI RESERVED, FMS & DOS
2A80 - 2BFF	UNUSED
2C00 - 2C7F	SAVED ATARI PAGE 0
2C80 - 2CFF	UNUSED
2D00 - 2D7F	PASCAL SYSTEM SAVED PAGE 0
2D80 - 2DFF	UNUSED
2E00 - 2EFF	PASCAL NON-PAGE 0 TEMPORARIES
2F00 - 2FFF	PASCAL EVALUATION STACK
3000 - 52FF	P-CODE EMULATOR
5300 - 53FF	OBJECT CODE HEADER FOR P-CODE FILES
5400 - 9BEF	PHASE 2 OF COMPILER (LARGEST PROGRAM)
9BF0 - AD00	COMPILER STACK AREA
AD00 - BC1F	COMPILER SYMBOL TABLE AREA (3872 BYTES)

AIABI Pascal

Release 0.0

Language Status

June 2, 1980

Table of Contents

- 1.0 Introduction
- 2.0 Pascal Standard
- 3.0 Atari Extensions

1.0 Introduction

This document describes the current release of the ATARI Pascal compilation system and its conformance to and exceptions with the proposed ISO standard and the ATARI Pascal extensions.

2.0 Pascal Standard

The eventual goal of the ATARI Pascal development project is to conform to the ISO/ANSI standard for Pascal. This standard is currently only a draft standard and has not been fully accepted.

Release 0.0 of ATARI Pascal does not implement the entire standard language. This is due to delivery time constraints and memory space. Future releases of the ATARI Pascal compiler will implement the entire language and modular compilation. The list below describes the standard features missing in Release 0.0:

1. Real data type, and all real functions.
2. NEW, DISPOSE procedures
3. SET data type
4. PAGE, PACK, UNPACK procedures
5. READ conversion for type INTEGER
6. ABS, SQR functions
7. EOF and EOLN for the console files
8. INPUT, OUTPUT pre-defined files
9. Procedure and Function names passed as Parameters
10. SUCC and PRED built-in functions

The choice for inclusion/exclusion from the Release 0.0 compiler of the above items was based upon delivery time and memory space (for the compiler) limitations.

ATARI Pascal contains (as of Release 0.0) only four major exceptions to the proposed standard. First, ATARI Pascal contains a "conventionalized" extension of the ELSE clause on the CASE statement. Second, ATARI Pascal allows the assignment of an integer value to a pointer variable. Third, ATARI Pascal allows the use of an optional file name parameter on the RESET and REWRITE statements to allow the user to select external files, by name, at run-time. Finally, ATARI Pascal allows integers to be used as pointers as in:

```
VAR
  I,J : INTEGER;

BEGIN
  I := 5000;
  J := I^;
```

This final exception was necessary to allow the ATARI Pascal compiler to be "bootstrapped" via a currently existing Pascal compiler and can be removed if desired.

3.0 Atari Extensions

ATARI Pascal contains two types of extensions to the standard language.

First, arrays of characters which begin at a lower bound of 0 are considered to be dynamic length strings. The array subscript 0 is the length of the data in the subscripts 1..n where n is the declared size. The length byte does not include the length byte itself (e.g. the length byte for 'ABCD' is 4). In the current release (0.0) there are only three operations which can be performed on strings: reading, writing (from the console and files) and their use as parameters to built-in procedures RESET, REWRITE and OPEN. Comparison and assignment of strings is not allowed. Movement of string data should use (for Release 0.0) the built-in MOVE procedure (see below).

Second, ATARI Pascal, Release 0.0, contains a number of "built-in" procedures which are conceptually in a library external to the compiler but as of the current release are actually built-in to the compiler as "magic" procedures. The user can redefine any of the "built-in" procedures as desired as these procedures, as specified in the standard (and Jensen and Wirth), exist in a scope surrounding the entire program.

ATARI Pascal 0.0 "Built-in" procedures:

1. MOVE(source_addr, dest_addr, length_in_bytes);

source_addr, dest_addr are either variable names or expressions; length_in_bytes is an integer expression.

2. OPEN(filevar, titlestring, result);

filevar is a variable of type FILE

titlestring : variable of type string

result : an integer VARIABLE to contain the IORESULT from the OPEN operation (see IORESULT for values)

3. BLOCKREAD, BLOCKWRITE (parameters below:)

(filevar, buffer_addr, result, relative_blk, numbytes);

filevar is a variable of type FILE with NO TYPE (e.g. F : FILE;)

buffer_addr is a VARIABLE name

result is an integer VARIABLE to hold the IORESULT from the I/O operation

relative_blk should be 0 (for Release 0.0 and current ATARI DOS)

numbytes is an integer expression specifying the size of the data to be transferred

4. CLOSE(filevar,result ((,KEEP) ; (,DELETE)));

filevar is a variable of type FILE

result is an INTEGER variable to contain the IORESULT of the close operation

Optionally the user may specify ,KEEP or ,DELETE on a CLOSE call. Note: The words KEEP and DELETE are NOT reserved words but MAGIC words when used in the CLOSE call. This means that the user may have variables of called KEEP and DELETE and this will NOT confuse the CLOSE procedure call.

The DEFAULT on a CLOSE operation is KEEP.

In order to DELETE a file the user must RESET the file and then CLOSE the file as the first operation after the RESET:

```
RESET(F,'....some_file_name....');  
CLOSE(F,RESULT,DELETE);
```

5. EXIT

The EXIT procedure performs a function exactly the same as the RETURN statement in BASIC. In actuality the code generated is simply a jump to the single RETURN opcode at the end of the procedure body. EXIT when used from the MAIN program will perform a WARMSTART operation (as specified by the ROM operating system).

6. BPT

DO NOT USE THIS! FOR EMULATOR DEBUGGING ONLY!

7. CHAIN(filevar)

filevar is an untyped FILE variable (e.g. F:FILE;)

The user must RESET the file containing the program to execute. The Emulator will then load in the file (using ATARI DOS load format) and begin execution at the address specified by the last header (this is \$5300 for compiler generated files). The user should guarantee that the loading of the program will not clobber the variable area in the program which is loading as this will result in a BAD IOCB error from the Emulator. EXECUTION BEGINS IN 6502 NATIVE OBJECT CODE MODE. Compiler produced programs contain a 16-byte 6502 machine code program preceeding the interpretive code which initializes the Emulator.

8. **INLINE(constant / constant / ... / constant);**

The **INLINE** procedure is used to place constants "in-line" in a **ATARI Pascal** program. This constant data may be character strings, integers, P-code or 6502 object code. Each constant in an **INLINE** statement is separated by a slash (/). If the user uses a named constant the compiler will generate two bytes of data. If the user uses a literal constant (e.g. \$0D) the compiler will generate either one or two bytes of data depending upon the size (considered as an integer) of the constant. If the literal constant can fit entirely into one byte then only one byte is generated otherwise two bytes are generated. This is particularly important to note for use with references to 6502 page 0.

Examples:

```
(* INSERTION OF P-CODE *)
INLINE( $FD / $0A / $FD / $0D ... );

(* INSERTION OF 6502 CODE *)
INLINE( $A9 / $53 / $A0 / $10 / $4C / $3200 );
(* THIS IS: LDA #$53
             LDY #$10
             JMP $3200 *)
```

The `INLINE` procedure "call" can also be used to create screen formats, etc. in a manner similar to the `BLOCK DATA` procedure in the FORTRAN language. The user can create a procedure which contains only constants and define a map for these constants using a `RECORD` variable. The user may then define a pointer to a variable of this type and assign the address of the constant data to this pointer variable. Then the constant data can be referenced from this pointer variable. An example is shown below:

```
PROGRAM DEMODATA;
```

```
TYPE
```

```
    PROMPTLINE = ARRAY [0..15] OF CHAR; (* STRING[14] *)
```

```
    PROMPTARR = ARRAY[0..3] OF PROMPTLINE;
```

```
VAR
```

```
    PLPTR : ^PROMPTARR;
```

```
PROCEDURE PROMPTDATA;
```

```
BEGIN
```

```
    INLINE(
```

```
        $OF / '123456789ABCDE' /
```

```
        $OF / 'ABCDEFGHJKLMN' /
```

```
        $OF / '!"$%&'()*_*=(')
```

```
    )
```

```
END;
```

```
BEGIN
```

```
    PLPTR := ORD(ADDR(PROMPTDATA))+4;
```

```
    (* see below for description of ADDR function *)
```

```
    (* 4 IS RELEASE 0.0 DEPENDENT *)
```

```
    WRITE(PLPTR^[2]); (* THIS WILL WRITE OUT STRING *)
```

```
END.
```

9. ADDR(variable):INTEGER;

The ADDR function will return the run-time memory address of a given variable. The variable may be fully qualified including subscripts and record fields, etc. This may then be used to set pointers.

10. IORESULT : INTEGER;

This function returns a code signifying the result of the last I/O operation as follows:

- 0 : no error
- 1 : OPEN error
- 2 : CLOSE error
- 3 : File READ error
- 4 : File WRITE error

11. SIZEOF(variable or type):INTEGER;

This functions returns the size of the specified variable or type in bytes. This is useful when using the MOVE procedure.

12. LO(expr) and HI(expr) : INTEGER;

LO and HI return the lower 8-bits and upper 8-bits respectively of the expression specified. The expression must not be a non-scalar or REAL.

13. GNB(filevar):CHAR;

GNB provides a high-speed character input access method for files. The user must declare the file var as:

filevar : FILE OF ARRAY [0..n] OF CHAR;

Where n is the desired buffer size. After the user has used RESET to open the file the user may then make calls to GNB and the Emulator will manage the reading of the data from the disk. Note: EOF is managed but the EOLN flag is not set or reset by GNB. When the end of the file is reached CHR(255) is returned and EOF(filevar) is set to TRUE. Subsequent calls to GNB will result in a value of CHR(255) being returned.

14. XIO(iocbnum,iocbrec):INTEGER;

XIO will generate a CIO call using the IOCB specified by IOCBNUM (note this is the number of the IOCB, NOT the number times 16). IOCBREC is a record describing the IOCB layout which has been filled in by the user. This record must contain 16 bytes (which means that it will, for now, contain a padding array of characters at the end). Fields in the IOCB which are one byte long must be declared as characters and fields which are two bytes long must be declared as integers/pointers. The value returned by XIO is the same as returned in the 6502 Y-register when calling CIO from assembly language.



AIAB1 Pascal

Release 0.0

Language User's Guide

June 2, 1980

Table of Contents

- 1.0 Source Program Preparation
- 2.0 Compiler control toggles
- 3.0 Compiler Informational Output
- 4.0 Compile-time errors

1.0 Source program preparation

Programs for the ATARI Pascal compiler are prepared using one of two text editors available on the ATARI 800 computer. Either the Assembler/Editor cartridge editor or the screen editor being developed by Atari may be used. Files suitable for input to the compiler may or may not have line numbers (line numbered and non-line numbered data may not be mixed in the same file). In Release 0.0 the user's program must end with the line '(**)'. This is a restriction which will be removed in a future release. The compiler will accept free form input as defined by the Pascal standard. Character string literals (e.g. 'abcd...') may not be longer than 80 characters and may not cross "physical" line boundaries (e.g. contain an EOL character).

2.0 Compiler control toggles

The Release 0.0 ATARI Pascal compiler contains two compiler control toggles which may be inserted into a source program by the user. A compiler control toggle is contained in a comment. IMMEDIATELY after the (*) the user places a \$ followed by a single letter (Release 0.0 supports I and Z). Following the letter the compiler expects at least one space and then the data to be used for the option. The data is terminated by recognition of the *) at the end of the comment.

The \$Z toggle allows the user to specify where in memory the stack frame allocation for variables, etc. will begin. The stack frames are allocated from this address DOWN towards low memory. The default is \$9800.

Example:

```
(*$Z $BC1F*) (* set stack to highest memory in a 48K system*)
```

The \$I toggle allows the user to include tokens from another file. The file should have already been tokenized using the S)can option of the monitor (see Pascal Operator's Guide). The user must specify a full legal file name for DOS/FMS to use. The extension on the file must be same extension as the token file specified by the S)can operation.

NOTE: THIS FEATURE IS STILL EXPERIMENTAL AND MAY CAUSE UNPREDICTABLE PROBLEMS.

Example:

```
PROGRAM TESTINCL;
```

```
(*$I D2:GLOBALS.TOK*) (* INCLUDE D2:GLOBALS.TOK HERE *)
```

```
BEGIN
```

```
  CH := 'X';    (* CH DEFINED IN GLOBALS FROM GLOBALS.TOK *)  
END.
```

3.0 Compiler Informational Output

When a program is compiled using the ATARI Pascal compiler the compiler (during Phase I and Phase II) generates information which is important to the user. This information is displayed on the screen during the compilation and is described below.

During Phase I the compiler will display the name of each procedure when it is encountered. Following this name are three numbers inside square brackets in the form:

[ll,pn,ma]

Where ll is the lex level of the procedure being compiled. The number 'pn' is the procedure number associated with that name. The main program is considered procedure 1 therefore the first internal procedure is considered procedure 2 and so forth. The number 'ma' is the number of bytes of memory available for symbol table space (in decimal). This number begins at approximately 3800 (in Release 0.0) and is used initially by the built-in procedure and function names leaving approximately 2900 bytes by the time the user's first symbol is encountered.

During Phase II the compiler will display the name of each procedure when the BODY of the procedure is encountered. Following this name are four numbers inside square brackets in the form:

[ll,pn,ad,ln]

Where ll and pn are as in Phase I the lex level and procedure number (in decimal). The number 'ad' the relative offset of the procedure body within the output code file of the compiler. This number 'ad' is in hexadecimal. The number 'ln' is the source program line number (starting with 1) of the beginning of the procedure body. NOTE: The line numbers used by the compiler bear no relationship to the line numbers in the file produced by the Assembler cartridge editor.

4.0 Compile-time errors

There are three programs which can produce error messages in the ATARI Pascal compiler system.

The syntax scanner will display an error number and an error message along with the last identifier scanned and up to 50 characters following the error. The syntax scanner for Release 0.0 will find the first error and then return to the monitor. This will be improved in future versions.

The symbol table builder (Phase I) will display error numbers and error messages and offer the user the option to continue or abort. Continuation is indicated by typing the space bar and aborting is indicated by typing the ESC key. Phase I will continue as long as possible.

The code generator (Phase II) will also display error numbers and error message and offer the user the option to continue or abort as described above.

If the user aborts from the syntax scanner (the only option) or aborts from the code generator (Phase II) the compiler will delete any output constructed up to that time and will close all files.

In the event of any error in Phase I, Phase II will not be executed.

When the compiler is aborted or terminated normally control will return to the monitor for further processing.

ATARI Pascal Emulator Interfacing

ATARI Pascal
Emulator Interface Guide

Release 0.0

June 2, 1980

Table of Contents

1.0	Introduction
2.0	General Information
3.0	Assembly language interface
3.1	Example of INLINE
3.2	Emulator memory map
3.3	Pascal memory allocation technique
3.4	Order of memory allocation

1.0 Introduction

The emulator is a psuedo 16 bit stack machine designed to execute Pascal programs in small micro computer environments. This emulator is not only effecient in its use of memory but is approxmitely 30% faster than UCSD Pascal on the 6502. This speed increase was made possible thourgh carefull design of the psuedo machine taking advantage of the 6502's addressing modes.

The emulator consists of 3 sections.

- a) System independent opcodes such as inteser arithmetic, loads, stores, compares, etc.
- b) Initialization and termination.
- c) System dependent Input and Output for console and files.

Each section is seperately assembled and linked together under the UCSD operating system to form a emulator which at this time is 2300 hex bytes long. When ROMing of the emulator becomes neccessary the I/O section will not be included but will be linked with the users program allowing the emulator to be less than 2000 hex bytes.

ARI Pascal Emulator Interfacing

2.0 General information

The emulator operates as follows:

- a) Fetch the next opcode pointed to by the PC
- b) Multiply the opcode by 2 and use as an index into the opcode translation table
- c) Jump to the routine pointed to by the opcode translation table
- d) Each opcode assumes the following information upon entry. First, the psuedo registers described above are valid. Second, the X register of the 6502 is the index into the parameter/evaluation stack and points to low byte of the 16 bit word on the stack.
- e) Each opcode exits by returning to the instruction fetch loop with register X of the 6502 as the parameter/evaluation stack index.

The psuedo machine architecture consists of 14 registers

- a) Program counter
- b) Return/variable stack for the procedure stack frames
- c) Parameter/evaluation stack for parameter passing and expression evaluation. This stack is 16 bits (2 bytes) wide, 256 bytes long (128 16 bit words), and constrained to a specific "page" with register X as the index to the top word of the stack. The words are PUSHed onto the stack by decrementing the X register and POPed by incrementing, also words are arranged on the stack with the standard low byte/high byte order used by the 6502. The evaluation stack is located at \$2F00 in Release 0.0 of the ATARI Pascal system.

The following example shows how items may be PUSHed and POPed:

PUSH:

DEX		;MAKE ROOM FOR HI BYTE
LDA	SOURCE	;GET HI BYTE
		; FOR ONE BYTE VALUES
		; THE HI BYTE MUST BE 0
STA	EVALPAGE,X	;PUSH THE HI BYTE
DEX		;MAKE ROOM FOR LO BYTE
LDA	SOURCE+1	;GET THE SOURCE LO BYTE
STA	EVALPAGE,X	;PUSH THE LOW BYTE

POP:

LDA	EVALPAGE,X	;GET THE LOW BYTE
STA	DEST	;STORE THE LOW BYTE

ARI Pascal Emulator Interfacing

```

      INX                ;DROP THE LOW BYTE
      LDA      EVALPAGE,X ;GET THE HI BYTE
      STA      DEST+1     ;STORE THE HI BYTE
      INX                ;DROP THE HI BYTE

```

- d) Local base register for accessing variables local to the presently executing procedure/function.
- e) 8 base (DISPLAY) registers which point to a procedure / function variable area. One for each active procedure / function. The following code PUSHes a 16 bit value off display register 3 with an index of 0235 hex:

DISP_REG_EXAMPLE:

```

      LDA      DR3                ;GET LOW BYTE OF DR3
      CLC
      ADC      #$35              ;ADD LOW BYTE OF INDEX
      STA      TMPBASE           ;STORE LOW BYTE
      LDA      DR3+1             ;GET HI BYTE OF DR3
      ADC      #$02              ;ADD HI BYTE OF INDEX
      STA      TMPBASE+1         ;STORE HI BYTE

      LDY      #1                ;POINT AT HI BYTE
      LDA      (TMPBASE),Y       ;GET HI BYTE
      DEX
      STA      EVALPAGE,X        ;PUSH HI BYTE
      DEY
      LDA      (TMPBASE),Y       ;GET LOW BYTE
      DEX
      STA      EVALPAGE,X        ;PUSH LOW BYTE

```

- f) LEXLEVEL register which is the lexlevel of the currently executing procedure/function
- g) TMPBASE temporary register used for address calculations

3.0 Assembly language interface

Assembly language may be used in Release 0.0 via the INLINE psuedo procedure. In future versions of the compiler the user is urged to not use this feature for assembly language but instead use the Assembler and Linker. But for now the INLINE feature may be used to execute 6502 assembly code. For a detailed description of INLINE refer to the ATARI Pascal Language Status section 3.8. Two additional operations are needed to actually execute assembly language routines. First there is a special emulator opcode (\$DF) which is used to start executing the native code. Second there is a jump instruction in page zero at location \$AD which must be "called" with a JSR instruction to return to the emulator.

3.1 Example of INLINE

```

FUNCTION MUL_BY_2(P:INTEGER):INTEGER;
  (*****
   PURP: MULTIPLY P * 2 AND RETURN IT
         ON THE EVALUATION STACK
   *****)

  CONST
    GOT0EM=$AD;      (* ADDRESS TO EXIT NATIVE CODE WITH *)

  BEGIN
    INLINE(
      (* START NATIVE CODE *)
      $DF/           (* EMULATOR OPCODE *)

      (* CALCULATE LOCATION OF P *)
      (* TMPBASE := LCLBASE + 10 *)
      $A9/10/        (* LDA #10 *)
      $18/           (* CLC *)
      $65/$C0/        (* ADC LCLBASE *)
      $85/$C8/        (* STA TMPBASE *)
      $A9/0/          (* LDA #0 *)
      $65/$C1/        (* ADC LCLBASE+1 *)
      $85/$C9/        (* STA TMPBASE+1 *)

      (* MULTIPLY P TIMES 2 *)
      $A0/0/          (* LDY #0 *)
      $B1/$C8/        (* LDA (TMPBASE),Y *)
      $0A/            (* ASL A ; LOW BYTE TIMES 2 *)
      $91/$C8/        (* STA (TMPBASE),Y *)
      $C8/            (* INY *)
      $B1/$C8/        (* LDA (TMPBASE),Y *)
      $2A/            (* ROR A ; HI BYTE TIMES 2 *)
      $91/$C8/        (* STA (TMPBASE),Y *)

      (* BACK TO EMULATOR *)
      $20/GOTOEM      (* JSR GOTO_EMULATOR *)
    );

    MUL_BY_2 := P;
  
```

ARI Pascal Emulator Interfacing

END;

ARI Pascal Emulator Interfacing

3.2 Emulator memory map

- a) all of the emulator addresses will change !!!
- b) all 16 bit words are in low byte high byte order
- c) <....> is what the user may do with that area

\$00..\$7F	Operating system variables <do not use>
\$80..\$87	Emulator permanent page zero vars <do not use>
\$88..\$AC	Emulator fetch loop <do not use>
\$9A..\$9B	PC psuedo Program counter <read only>
\$AD..\$AF	Jump instruction for returning to emulator from native code
\$B0..\$BF	DR0..DR7 8 16 bit display registers DR0 = \$B0 .. DR7 = \$BE <read only>
\$C0	LCLBASE local base register address of the presently executing procedure/functions local variables <read only>
\$C2	PRGSP return/variable stack pointer <read only>
\$C4	EVALSP evaluation/parameter stack (byte) used to save reg X in various opcodes. <read/write temporary>
\$C6	LEXLEVEL (byte) of present executing procedure/function <read only>
\$C8	TMPBASE used for address calculations <read/write temporary>
\$D0..\$FF	Temporaries used by emulator the user my use them with caution <read/write temporary>
\$100..\$2BFF	ATARI -- FMS, DOS, BUFFERS <????>

ARI Pascal Emulator Interfacing

\$2C00..\$2EFF	Emulator buffers <do not use>
\$2F00..\$2FFF	Emulator parameter/evaluation stack <read/write via reg X>
\$3000..\$52FF	Emulator code area <do not use>
\$5300..top of memory	Pascal code and variable area

ARI Pascal Emulator Interfacing

3.3 Pascal memory allocation technique

3.3.1 Size of various types

type ----	example -----	size ----
char	ch:char;	1 byte
integer	i:integer;	2 bytes
boolean	b:boolean	2 bytes
enumerated	color:(red,green,blue);	2 bytes
subrange	'a'..'z'	1 byte
	0..255	2 bytes
arrays	array[1..20] of integer	40 bytes
	array[0..10] of char	11 bytes
records	record i:integer; ch:char; strg:array[0..6] of char; end;	sum of the field sizes 10 bytes here
text	f:text	33 bytes
file of something	f:file of array[0..1023] of char;	33 bytes + size of something
case variant records	record case boolean of true:(c:char); false:(j:integer); end;	sum of size of largest case 2 bytes here

3.4 Order of memory allocation

In general variables are allocated in the order they are processed by the compiler. The trick is to know the order of processing. There are 2 cases: VAR declaration and parameters declaration.

a) PARAMETERS

Parameters are allocated first and in the reverse order. VAR parameters are allocated 2 bytes for the address and value parameters are allocated the size of its type, except for value arrays and records parameters which are allocated an additional 2 bytes following the array or record. For procedures variables start at offset 0, but for FUNCTIONS they start at offset 10 and the function value is located at offset 0.

a) VARs

All singularly defined variables are allocated in order given, and variables defined in lists are allocated in the reverse order. VARs are allocated memory after parameters.

PROGRAM ALLOCATION:

TYPE

```
SOME_RECORD: RECORD
    I: INTEGER;
END;
```

VAR

```
J: INTEGER;          (* DRO OFFSET = 0 *)
K: CHAR;              (* DRO OFFSET = 2 *)
```

```
FUNCTION TEST(P1: INTEGER; VAR P2, P3, P4: CHAR;
    VAR F: FILE; P5: SOME_RECORD): INTEGER;
```

```
(*****
THIS PRECEDURE WILL USE
DISPLAY REGISTER DR1 FOR
POINTING TO ITS VARIABLES
```

```
P5 = DR1 OFFSET 10
F  = DR1 OFFSET 14
P4 = DR1 OFFSET 46
P3 = DR1 OFFSET 48
P2 = DR1 OFFSET 50
P1 = DR1 OFFSET 52
```

```
*****)
```

VAR

```
I: INTEGER;          (* I = DR1 OFFSET 54 *)
C: CHAR;              (* C = DR1 OFFSET 56 *)
```

ARI Pascal Emulator Interfacing

```
A:ARRAY[1..3] OF CHAR; (* A = DR1 OFFSET 57 *)
K:BOOLEAN;              (* K = DR1 OFFSET 59 *)
L,M,N:INTEGER;          (* N = DR1 OFFSET 61 *)
                        (* M = DR1 OFFSET 63 *)
                        (* L = DR1 OFFSET 65 *)

R:RECORD
  I:INTEGER;             (* R.I = DR1 OFFSET 67 *)
  C:CHAR;                (* R.C = DR1 OFFSET 69 *)
  CASE B:BOOELAN OF
    TRUE:(K:INTEGER);    (* R.K = DR1 OFFSET 72 *)
    FALSE:(CH:CHAR);     (* R.CH= DR1 OFFSET 72 *)
  END;
```

```
BEGIN
END;
```

MT MicroSYSTEMS

Specializing In Innovative
Information Processing

```
(*)-----*)
(* COMPUTE ONLY BENCHMARK *)
(* COURTESY OF THE      *)
(* UNIX USER'S GROUP VIA *)
(* JIM GILBREATH, MANAGER *)
(* TECHNICAL SERVICES AT *)
(* NAVAL OCEAN SYSTEMS CTR*)
(*)-----*)
PROGRAM PRIMES;
CONST
    SIZE = 8190;
VAR
    I,J,K : INTEGER;
    COUNT : INTEGER;
    PRIME : ARRAY[0..SIZE] OF BOOLEAN;
BEGIN
    WRITELN('Type <return> to start');
    READLN;
    COUNT := 0;
    FOR I := 0 TO SIZE DO
        PRIME[I] := TRUE;
    FOR I := 0 TO SIZE DO
        IF PRIME[I] THEN
            BEGIN
                J := I+1+3;
                K := I + J;
                WHILE K <= SIZE DO
                    BEGIN
                        PRIME[K] := FALSE;
                        K := K + J;
                    END;
                COUNT := COUNT + 1;
            END;
        WRITELN(COUNT, ' primes');
        READLN;
    END;
    (**)
```

MT MicroSYSTEMS

Specializing In Innovative
Information Processing

```
10 PROGRAM TESTGET;
20 VAR
30   F : TEXT;
40   CH: CHAR;
50   I : INTEGER;
60 BEGIN
70   RESET(F, 'D:ERRORS.TXT');
80   WHILE NOT(EOF(F)) DO
90     BEGIN
100      WHILE NOT(EOLN(F)) DO
110        BEGIN
120          CH := F^;
121          WRITE(CH);
130          GET(F)
140        END;
150        WRITELN;
160        READLN(F)
170      END;
180      WRITELN('END OF FILE');
190      READLN
2000 END.
```

MT MicroSYSTEMS

Specializing In Innovative
Information Processing

```
0100 PROGRAM TESTROL;  
0110 VAR  
0120   F : TEXT;  
0130   CH: CHAR;  
0140   I : INTEGER;  
0141   S : PACKED ARRAY [0..80] OF CHAR;  
  
0150 BEGIN  
0160   RESET(F, 'D:ERRORS.TXT');  
0170   WHILE NOT(EOF(F)) DO  
0180     BEGIN  
0181       READLN(F,S);  
0182       WRITELN(S);  
0190     END;  
0200   WRITELN('END OF FILE');  
0210   READLN  
0220 END.
```

MT MicroSYSTEMS

Specializing In Innovative
Information Processing

```
0100 PROGRAM TESTGNB;  
0110 VAR  
0120   F : FILE OF ARRAYED...1023 OF CHAR;  
0130   CH: CHAR;  
0140   I : INTEGER;  
0150 BEGIN  
0160   RESET(F, 'D:ERRORS.TXT');  
0170   WHILE NOT(EOF(F)) DO  
0180     BEGIN  
0190       CH := GNB(F);  
0200       WRITE(CH);  
0230     END;  
0240   WRITELN('END OF FILE');  
0250   READLN  
0260 END.
```